

This homework is due Friday, October 3, by 4:00pm. Please place it in the box marked “CSc 545” in the Department mailroom, Gould-Simpson 713. The questions are drawn from the material in the lectures on finding the k th smallest and dynamic programming.

The homework is worth a total of 100 points. When questions with several parts do not specify the points for each part, each part has equal weight.

For questions that ask you to design an algorithm that runs in a given time bound, you must:

- (i) describe an algorithm for the problem using *prose* and *pictures*,
- (ii) argue that your algorithm is correct, and
- (iii) analyze the running time of your algorithm.

Providing pseudocode is not necessary, but do explain the *ideas* behind your algorithm and its correctness. Only give high-level pseudocode if it aids the time analysis of your algorithm.

For questions that ask you to design a *dynamic programming* algorithm, remember to use the four-part framework:

- (1) *characterize* the recursive structure of an optimal solution,
- (2) *derive* a recurrence equation for the value of an optimal solution,
- (3) *evaluate* the recurrence bottom-up in a table, and
- (4) *recover* an optimal solution from the table of solution values.

You will be graded on each part. Be sure to analyze the time for Parts (3) and (4) of your algorithm.

Remember to write on just one side of a page, do not use scrap paper, put your answers in the correct order, and staple your pages together. If you can't solve a problem, state this, and write only what you know to be correct. Neatness and conciseness count.

- (1) **(Finding quantiles)** (10 points) For a set S of n numbers and an integer k where $1 \leq k \leq n$, the k th quantiles of S are $k - 1$ elements from S whose ranks in S divide the sorted set into k groups that are of equal size to within one unit.

Given an unsorted array A of n distinct numbers, design an algorithm that finds the k th quantiles of A in $O(n \log k)$ time.

(Note: To help explain the problem, the 4-quantiles of a set of scores are the values that define the 25-, 50-, and 75-percentile cutoffs. Similarly, the 10-quantiles of a set are the values that define the 10-, 20-, 30-, ..., and 90-percentile cutoffs.)

- (2) **(Finding elements near the median)** (10 points) Given an unsorted array A of n distinct numbers and an integer k where $1 \leq k \leq n$, design an algorithm that finds the k numbers in A that are closest in value to the median in $\Theta(n)$ time.

- (3) **(Finding the k th smallest in the merge of sorted arrays)** (20 points) Given two sorted arrays $A[1 : m]$ and $B[1 : n]$ of all distinct numbers, and an integer k where $1 \leq k \leq m + n$, design an algorithm that finds the k th smallest element in the *merge* of arrays A and B in $O(\log k)$ time.

(Note: Explicitly merging A and B will take $\Theta(m + n)$ time, which is too costly.)

- (4) **(Editing strings)** (30 points) Given two strings $A[1 : m]$ and $B[1 : n]$, the *edit distance* between A and B is the minimum cost of a script that edits A into B . A script is a series of edit operations, each edit operation has a non-negative cost, and the cost of a script is the sum of the costs of its operations.

The allowed edit operations in a script are:

- *copy*, which leaves a character unchanged, and has cost 0,
- *substitute*, which replaces a character a with another character b , and has cost c_{sub} ,
- *insert*, which adds a character a into a string, and has cost c_{ins} ,
- *delete*, which removes a character a from a string, and has cost c_{del} , and
- *transpose*, which replaces two adjacent characters ab in a string by the characters ba , and has cost c_{tra} .

Design a *dynamic programming* algorithm to compute the edit distance between A and B and recover the corresponding edit script in $\Theta(mn)$ time. You may assume that an optimal script never edits a given character more than once. The costs of operations are part of the input to your algorithm.

(Hint: Order the operations in an edit script so they occur left-to-right across string A , and then examine the possible ways in which an optimal script could end.)

- (5) **(Restricted traveling salesperson)** (30 points) Given a set S of points in the two-dimensional (x, y) plane, the *traveling salesperson problem* is to find a closed polygon whose vertices are the points in S and that minimizes the perimeter of the polygon, in other words, the sum of the lengths of the edges in the polygon. Such a polygon is a series of line segments between points in S that touches every point in S exactly *once*, and that returns to the point at which it began.

To simplify the problem, we will *restrict* the polygons that we consider. Assume the points in S are indexed p_1, p_2, \dots, p_n , sorted according to increasing x -coordinate, and that their x -coordinates are distinct. (So for $p_i = (x_i, y_i)$, we have $x_1 < \dots < x_n$.) We only consider restricted polygons that:

- (1) starting from p_1 , follow a series of line segments to points that are *increasing* in x -coordinate, until reaching p_n , and then
- (2) continuing from p_n , follow a series of line segments to points that are *decreasing* in x -coordinate, until returning to p_1 .

In other words, in walking around the exterior of such a restricted polygon from vertex p_1 , the vertices first form a left-to-right sequence in which x -coordinates are increasing, and then form a right-to-left sequence in which x -coordinates are decreasing.

Design a dynamic programming algorithm for this *restricted form* of the traveling salesperson problem that finds an optimal restricted polygon in $O(n^2)$ time, given a set S with n points.

(Note: It may help you to represent such a polygon as a binary string $s_2s_3 \dots s_{n-1}$, where character s_i is either L or R depending on whether point (x_i, y_i) is on the “left-to-right” or “right-to-left” portions of the polygon.)

(Hint: In working out the structure of an optimal solution, ask how an optimal solution ends. Note that on removing the end of a solution, what remains is no longer a closed polygon. So the form of the recursive subproblem that you solve may be different than the form of the original problem. Also, you may need to develop recurrences for *two* quantities that depend on each other.)